# DALB-MC: A Novel Dynamic Adaptive Loading Balance Algorithm to Improve the Computational Performance of Monte Carlo Codes

Jiajun WU[1,2], Hui ZHANG[1,2], Ankang HU[1,2], Yanheng PU[1,2], Zhen WU[1,2,3*], Rui QIU[1,2], Junli LI[1,2]

[1] *Department of Engineering Physics, Tsinghua University, Beijing 100084, China*
[2] *Key Laboratory of Particle and Radiation Imaging of Ministry of Education, Beijing 100084, China*
[3] *Nuctech Company Limited, Beijing 100084, China*

**Abstract:** This paper introduces a novel algorithm aimed at enhancing the computational performance of Monte Carlo codes: A Dynamic Adaptive Load Balancing Algorithm for Monte Carlo Codes (DALB-MC algorithm). With the widespread use of multicore processors, some Monte Carlo codes have been adapted to leverage both local thread parallelism and global parallelism through message passing between nodes, enabling concurrent execution of independent tasks and efficient merging of results. However, as shielding engineering problems increase in complexity and computational environments become more heterogeneous, traditional parallel computing algorithms face limitations in handling Monte Carlo simulations, particularly in ensuring adapting to diverse hardware resources. The proposed DALB-MC algorithm addresses these challenges by dynamically partitioning tasks into multiple sub-tasks and allocating them in real-time based on the computational capacity of each node, thereby optimizing resource utilization and reducing overall simulation time. In this study, we implement DALB-MC algorithm on MCShield and demonstrate the correctness and effectiveness of the algorithm in improving the performance of the Monte Carlo code through self-defined arithmetic examples. The experimental results show that compared with original algorithm, DALB-MC algorithm achieves about 20% computation time acceleration in massively parallel computation and significantly improves the overall efficiency of the program. In addition, a Linux supercomputing platform is used to test a complex benchmark case to further verify the optimization effect of DALB-MC algorithm under the computational scale of real engineering applications.

**Keywords:** Monte Carlo simulation, Parallel computing efficiency, Load balancing algorithm, MCShield, Multicore processors.

* Corresponding author (Email: wuzhen97@tsinghua.org.cn)

# 1 Introduction

With the advent of multicore processors in the 2000s, some Monte Carlo codes were adapted to utilize threaded parallelism through MPI, OpenMP[1], or vendor-specific pragmas, marking a significant milestone in the development of Monte Carlo code[2]. Monte Carlo parallel computing combines local parallelism using threading on multicore processors with global parallelism through message-passing between nodes, allowing independent jobs to run concurrently and results to be combined. [3] Current high-performance computing systems support up to 96 cores per node (AMD EPYC 9654 processor), with future developments expected to increase the number of cores and threads per node. To maximize the capabilities of today's large-scale high-performance computing systems, numerous Monte Carlo codes are being developed to fully leverage modern architectures. Such codes as Geant[11], MCNP[12], OpenMC[13], FLUKA[14], KENO[15], EGS[16], and Serpent[17], and many others are newer and more readily adapted to large-scale applications. Among them, MCShield[18], developed by the Radiation Protection and Environmental Protection Laboratory at Tsinghua University, is a Large-scale Monte Carlo program specifically designed for radiation shielding calculations involving coupled neutron, photon, and electron transport, tailored to meet diverse shielding challenges.

The Monte Carlo parallel computation process usually follows the Master-Slave Model[23]: after completing the definition of the source terms, geometric and physical processes, during the initialization phase, the master process accepts user-defined parameters, creates a specified number of slave processes and assigns computational tasks to them. A seed sequence of random numbers is created prior to the task assignment to ensure simulation independence between different slave processes, which is sent by the master process to each slave process as one of the initialization data. Efficient resource scheduling and utilization can be achieved by developing load balancing algorithms.

The concept of 'load balancing' was first proposed by shiny et al[4] in order to achieve an increase in resource utilization and a reduction in the failure rate of data center nodes in cloud computing problems. After the development in recent years, load balancing algorithms can be mainly classified into two types of algorithms: static load balancing and dynamic load balancing. Static load balancing algorithms, such as Weighted Round Robin[5], allocate the load of the system in a one-time basis based on a priori knowledge of the static system state, attributes, and capabilities, including information such as memory, storage capacity, and processing power, etc., and such algorithms don't take into account the change of load at runtime[6]. Dynamic load balancing algorithms, like Adaptive loading balance[10], are different from static ones in that they are designed based on dynamic environments and take into account the previous state of the system and flexibly adjust the load of each node during the computation process. This increases the computational overhead of the system and is more complex than the static algorithm due to the need for additional storage of the system's previous state[6,7].

As the nuclear industry continues to expand, shielding calculation problems are becoming increasingly complex. Large-scale Monte Carlo codes, which involve tracking vast numbers of particles, typically require thousands of threads to process data in parallel. However, the heterogeneity of computational resources introduces significant challenges to the performance of parallel computing. Traditional static load-balancing methods commonly employed in Monte Carlo codes are beginning to reveal their limitations. High-performance computing nodes used in

Monte Carlo simulations often vary in hardware specifications, including CPU speeds, core counts, available memory, disk I/O capacity, and network bandwidth. These differences can lead to inefficiencies, as computational performance tends to degrade when the number of threads increases. To maintain efficiency, effective load balancing must account for these hardware disparities, ensuring that less capable nodes are not overloaded while fully leveraging the capabilities of more powerful ones.

In parallel computing, optimizing the use of computational resources to enhance the performance of Monte Carlo codes is essential. These challenges not only hinder overall system performance but also restrict the applicability of parallel computing in large-scale radiation shielding scenarios. This paper introduces a Dynamic Adaptive Load Balancing algorithm for Monte Carlo codes (DALB-MC), which will be implemented for the first time in a Monte Carlo code, utilizing MCShield as the platform for this innovative technology.

# 2 Material and methods

## 2.1 The Parallelization Framework of MCShield

The parallel functionality of MCShield relies on the Message Passing Interface (MPI), a standard communication protocol and library crafted to facilitate communication and collaboration among multiple processes in a parallel computing environment. Various open-source implementations of MPI standard exist, including Open MPI[19,20] and MPICH[21,22], among others. In MCShield, MPI Master-Slave model is employed. The master process is responsible for assigning tasks to slave processes, which execute the tasks independently and return the results to the master upon completion. The master handles task distribution, result collection, and overall control logic, while slave processes focus on the specific computational tasks assigned to them. As illustrated in Figure 2.1, each particle event is simulated by a series of slave processes, with the master process guiding and managing the simulation. Using MPI messaging functions, each slave process receives initialization data from the master process to commence computation. To optimize memory usage, geometry and physical tables are shared among slave processes, whereas the specific routing and stepping processes remain private to each individual slave process. Once slave processes complete their assigned tasks, they utilize message-passing functions to send their results back to the master process, which then consolidates and outputs the final results.

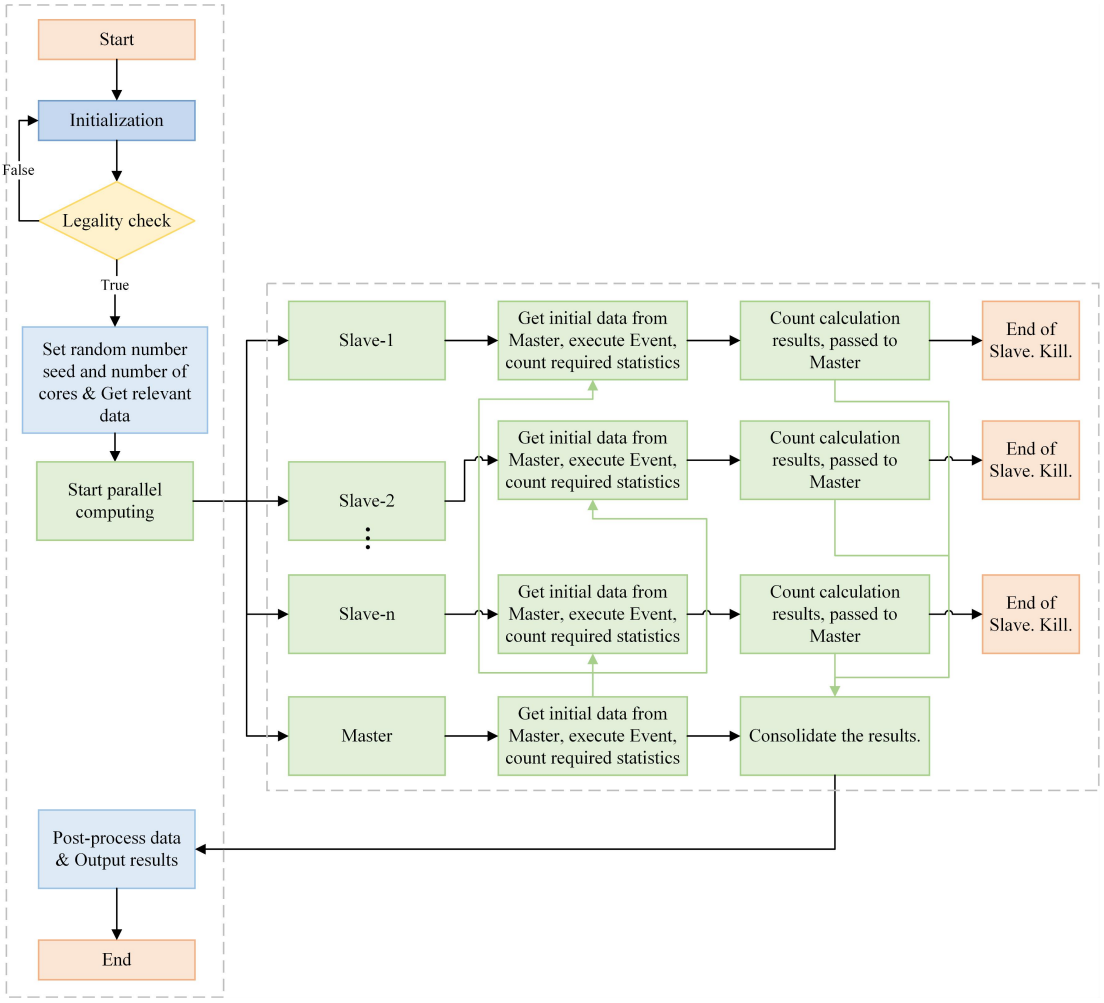


Figure 2.1 The Parallelization Framework of MCShield

## 2.2 The Dynamic Adaptive Loading Balance Algorithm for MCShield

This paper addresses the issues of wasted performance and inefficient parallelism in massively parallel particle transport simulations by proposing a Dynamic Adaptive Load Balancing algorithm for Monte Carlo codes (DALB-MC). Unlike static load balancing algorithms, which allocate the computation task at once, DALB-MC algorithm divides the task into multiple discrete subtasks based on a small initial quota. At the start of the simulation, subtasks are distributed from the master process to each slave process. Each process completes its assigned subtasks and then requests new ones until all are executed. Each subtask maintains an equal number of simulated particles, allowing users to define the particle count of the subtask based on the total number of particles and the number of parallel cores, typically set to 1000 particles per subtask.

The comparison schematic illustrating the performance before and after implementing DALB-MC algorithm is shown in Figure 2.2. As depicted, without DALB-MC algorithm, when a process(worker) experiences prolonged simulation times—often due to lengthy particle histories or weaker CPU power on certain nodes—many other processes may complete their tasks and become idle. This results in significant CPU resource wastage and extends total simulation time due to a few lagging processes. In contrast, DALB-MC algorithm enhances efficiency by allowing each slave process to request a new subtask immediately upon completing its current subtask. This means that, despite variations in execution speed among processes, none enter an idle state during

the simulation. Faster processes will execute more subtasks, while slower ones will execute fewer, ensuring that all available computational resources are utilized effectively. This new algorithm significantly improves resource utilization and parallel efficiency, ultimately reducing total simulation time.


Figure 2.2 Schematic Illustration of Performance Metrics Before and After DALB-MC algorithm Implementation

Left：Static load balancing algorithm; Right: DALB-MC algorithm

The framework of DALB-MC algorithm includes the initialization of MPI, the branching of the master-slave logic and the end of MPI. The flowchart of DALB-MC algorithm is shown in Figure 2.3, where the main steps are as follows:

***(1) Initialization of master and slave processes***

Before parallel computation begins, the program first initializes or updates private variables for the master process and all slave processes(workers). The master process must record information such as the total number of particles, the number of subtasks, the number of particles per subtask, the number of completed subtasks, the number of particles simulated, Master process ID, the number of available slave processes, and the number of terminated processes. Meanwhile, each slave process keeps track of information such as number of particles simulated, number of tasks completed, particles in current task, and work status (0: awaiting task assignment, 1: task in progress, 2: all tasks completed).

***(2) Initial task assignment***

When parallel computation starts, the master process assigns the first task to itself and all slave processes, which then enter a working state upon receiving their initial tasks. Task allocation and reception is facilitated by a message-passing interface, which handles the transmission of information such as particle counts and process states. This allows the master process to communicate effectively with each slave process.

***(3) Task computation***

Once each process has received its initial task, it begins to compute its respective subtasks. After the master process completes its first subtask, it enters a listening mode and monitors the number of tasks completed, the particle count, and the status of each slave process. As each worker completes its current task, the master process assesses the overall task completion status and either assigns a new task or terminates that worker's tasks. After completing its current task, each slave process notifies the master process and requests the next task, and this loop continues until all subtasks are completed and slave processes are terminated.

***(4) Completion of parallel computation***

When all subtasks have been completed and all slave processes have terminated, the simulation task is completed. Each slave process updates its task statistics, and the master process

consolidates the results from all workers, updating parameters such as the total number of particles simulated (NPS). The master process then determines whether another simulation phase is required. When all particles have been fully simulated, the parallel computation ends and the program proceeds to the result output phase and other subsequent steps.

In this study, DALB-MC algorithm is implemented in the simulation control class within MCShield, following the above task allocation logic. Interfaces related to MPI parallel management and task distribution are encapsulated within a parallel management class, which provides several functions to support parallel management and task distribution functionalities.

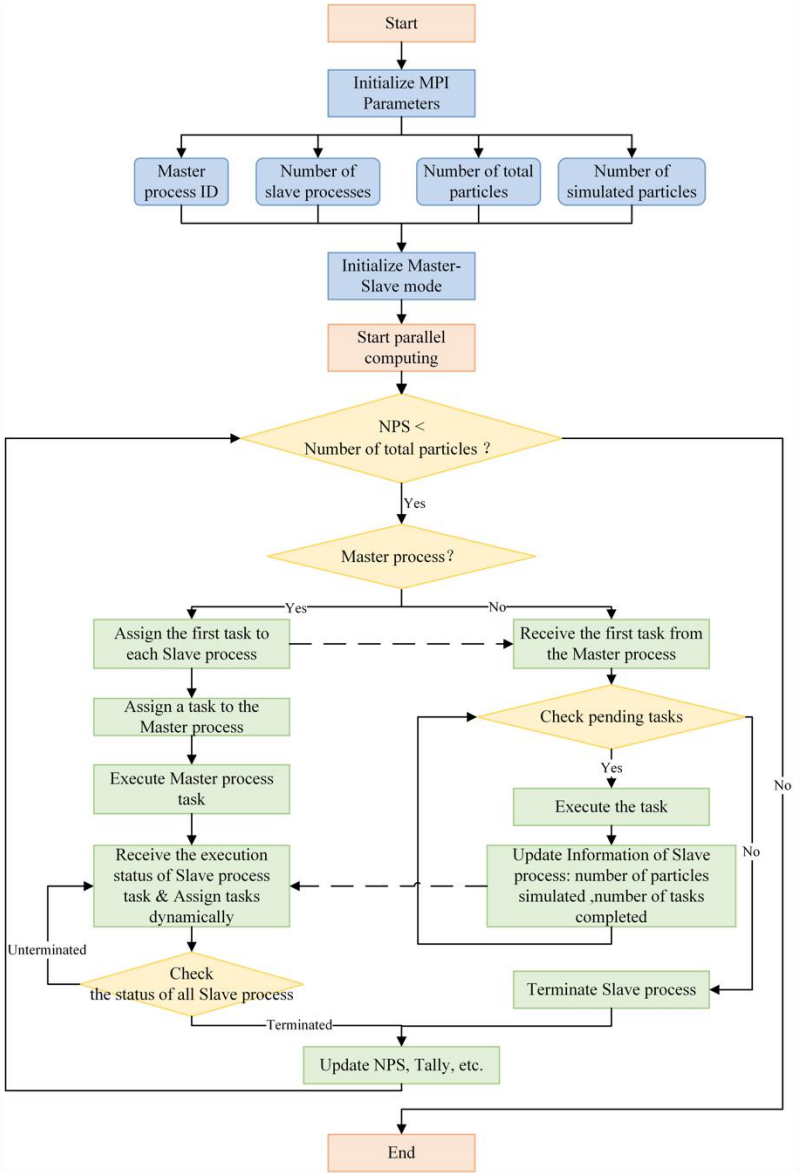


Figure 2.3 Flowchart of DALB-MC algorithm applied to MCShield

## 2.3 Experimental setups and platforms

### 2.3.1 Simplified reactor shielding model

The correctness and effectiveness of DALB-MC algorithm are tested using a self-defined simplified model in small-scale Windows clusters. The geometry of the test case is a simplified reactor shielding model, with three views of the specific geometric structure shown in Figure 2.4.

The core is located in the bottom center of the silo and is externally shielded by a double layer of water and concrete. The material inside the silo space is air, and outside is vacuum. Two air ducts, each with a radius of 0.1 m, are distributed along the X-axis and Z-axis directions within the reactor, with their specific locations illustrated in Figure 2.4. The source term is a body source uniformly sampled within the core geometry, featuring a monoenergetic energy of 14.1 MeV and isotropic sampling angles. The statistical region is defined as the cylindrical body flux at the top of the axial air duct on the right side of the core, with the statistical cylinder having a radius of 0.1 m and a height of 0.1 m. The correctness of the algorithm is verified by comparing the Monte Carlo simulation results (statistical regional body flux results) applying DALB-MC and original algorithm.

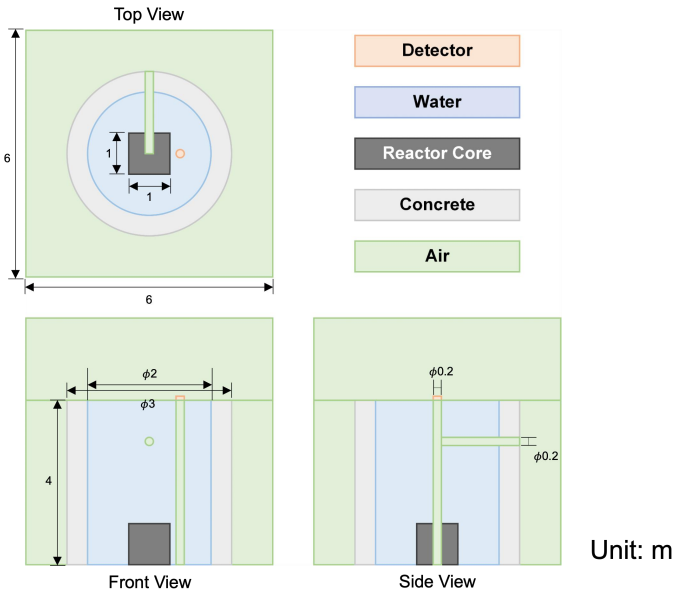


Figure 2.4 Three views of the self-defined simplified reactor shielding model

### 2.3.2 Complex reactor shielding model

VENUS benchmark model is tested using a supercomputing Linux platform to simulate node performance differences and potential communication issues that may be encountered in real application scenarios, thereby evaluating the optimization effect of DALB-MC algorithm. VENUS Critical Unit is a zero-power reactor located in Belgium. It has been modified to provide a large amount of baseline measurements and Monte Carlo data for particle cross sections and particle transport procedures. VENUS-III was the third experiment conducted in 1988. The quarter core plus reflective surfaces model is constructed according to VENUS-III (Figure 2.5). The effectiveness of the new algorithm in enhancing Monte Carlo code performance is assessed using Figure of Merit (FOM) as a performance evaluation reference metric.

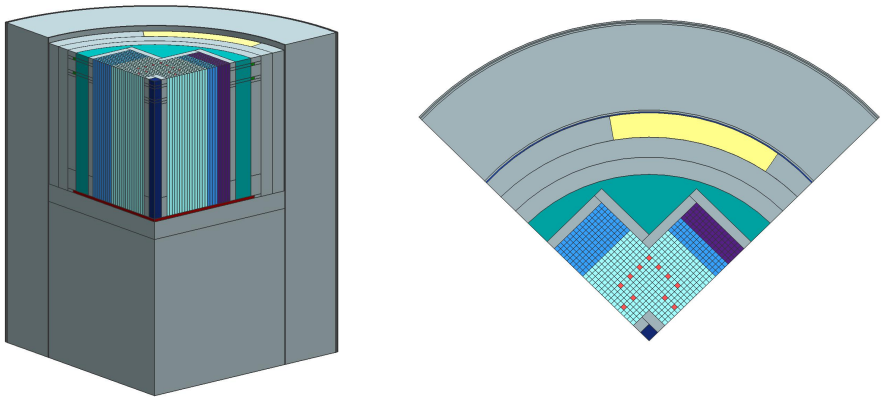

Figure 2.5 Schematic geometry of VENUS benchmark model

### 2.3.3 Computing platforms

Small-scale Windows clusters consists of 8 nodes, each with 100 cores for parallel computing, while the supercomputing Linux platform employs Beijing Supercomputing BSCC-A2 to increase the number of computing nodes and parallel cores for more extensive testing. The specific configuration of the computing platform is detailed in Table 2.1.

Table 2.1 Configuration information for two computing platforms

| **Computing platform** | **Operating system** | **Number of CPU nodes** | **Single-node CPU configuration** | **Number of cores per node** | **Single node memory** | **Total number of cores** |
|---|---|---|---|---|---|---|
| Small-scale Windows Clusters | Windows | 8 (4+4) | Intel Xeon 6240R Processor | 24 | 128GB | 100 |
| | | | Intel Xeon Gold 6148 Processor | 24 | 128GB | |
| Supercomputing Linux Platform (Beijing Supercomputer BSCC-A2) | Linux | 40 | AMD EPYC 7452 Processor | 64 | 256GB | 2560 |

# 3 Results and Discussion

## 3.1 Verification of Algorithm Correctness

Two independent Monte Carlo simulations were conducted using DALB-MC algorithm and the static load balancing algorithm (original algorithm) on the small-scale Windows cluster. The total number of simulated particles in the test was $10^8$, with results output after every $10^7$ particles simulated. The results of the Monte Carlo simulations include body flux results and error of the statistical region, and the trend of related deviation between the two statistical results as the total number of simulated particles (NPS) increases. These findings are summarized in Table 3.1 and illustrated in Figure 3.1.

Table 3.1 Comparison of Monte Carlo simulation results of two algorithms

| Number of simulated particles | Original algorithm | | DALB-MC algorithm | | Related deviation |
|---|---|---|---|---|---|
| | Flux (1/cm²) | Error | Flux (1/cm²) | Error | |
| 1.0E+07 | 5.95E-03 | 31.15% | 4.90E-03 | 34.61% | -17.66% |
| 2.0E+07 | 5.58E-03 | 22.63% | 4.75E-03 | 24.00% | -14.79% |
| 3.0E+07 | 5.76E-03 | 18.26% | 5.62E-03 | 18.25% | -2.33% |
| 4.0E+07 | 5.54E-03 | 16.44% | 5.08E-03 | 16.52% | -8.35% |
| 5.0E+07 | 5.13E-03 | 14.99% | 4.63E-03 | 15.29% | -9.72% |
| 6.0E+07 | 4.82E-03 | 13.84% | 4.29E-03 | 14.15% | -10.87% |
| 7.0E+07 | 4.51E-03 | 13.11% | 4.42E-03 | 12.94% | -1.83% |
| 8.0E+07 | 4.69E-03 | 11.98% | 4.67E-03 | 11.98% | -0.41% |
| 9.0E+07 | 4.90E-03 | 11.19% | 4.80E-03 | 11.22% | -1.98% |
| 1.0E+08 | 5.04E-03 | 10.48% | 4.84E-03 | 10.52% | -3.99% |

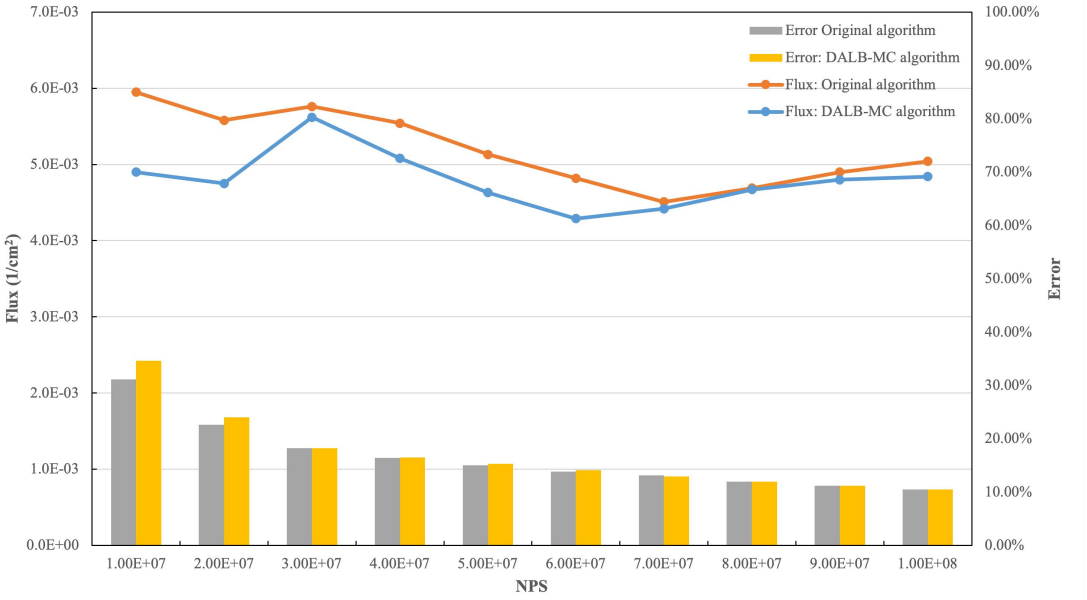


Figure 3.1 Trends in statistics with NPS for Monte Carlo simulations of two algorithms

From the above results, it is evident that the outcomes of the two simulations, static and dynamic, are in good agreement. The relative deviation of flux in the statistical region at the end of the all-particle simulation is -3.99%, falling within the acceptable range of the statistical error mainly caused by random numbers. Additionally, the convergence trends of the statistical error with the increasing particles are similar, confirming the correctness of DALB-MC algorithm.

## 3.2 Verification of Algorithm Effectiveness

For further performance comparison, the computation times of two simulations, one implementing original algorithm and the other DALB-MC algorithm, are analyzed in this study. The Computational Advantage Factor, or Figure of Merit (FOM), is used as a reference index for performance evaluation, calculated as follows:

$$FOM = \frac{1}{R^2 T} \tag{1}$$

Where $R$ is the statistical error and $T$ is the computational time, FOM value allows for a comprehensive measurement of the program's computational performance. The number of subtasks completed by all worker processes at the end of the simulation procedure using DALB-MC algorithm is presented in Figure 3.2. The cumulative computation time and FOM value before and after DALB-MC algorithm implementation with NPS are detailed in Table 3.2 while the trend curves of computation time and FOM value with NPS for the two simulations are illustrated in Figure 3.3 and Figure 3.4.

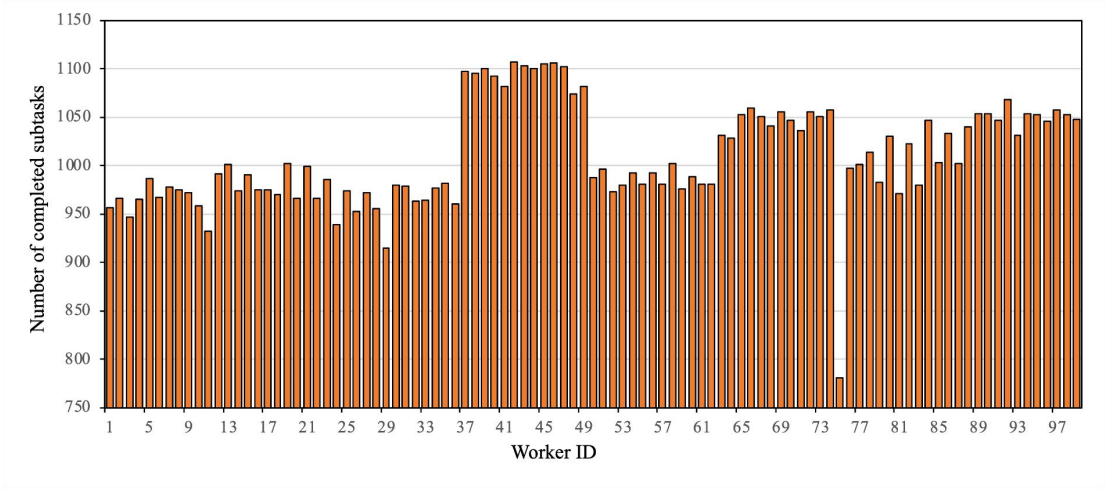


Figure 3.2 Comparison of the number of tasks completed by each core for MCShield applying DALB-MC algorithm

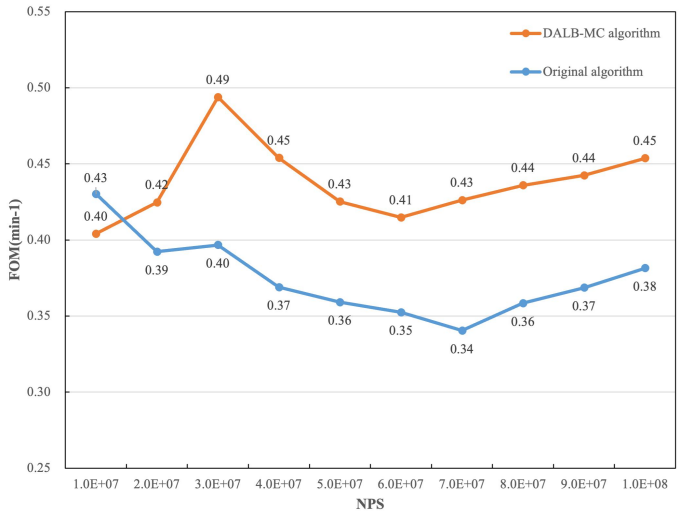


Figure 3.3 Comparison of the simulation FOM of MCShield applying two algorithms

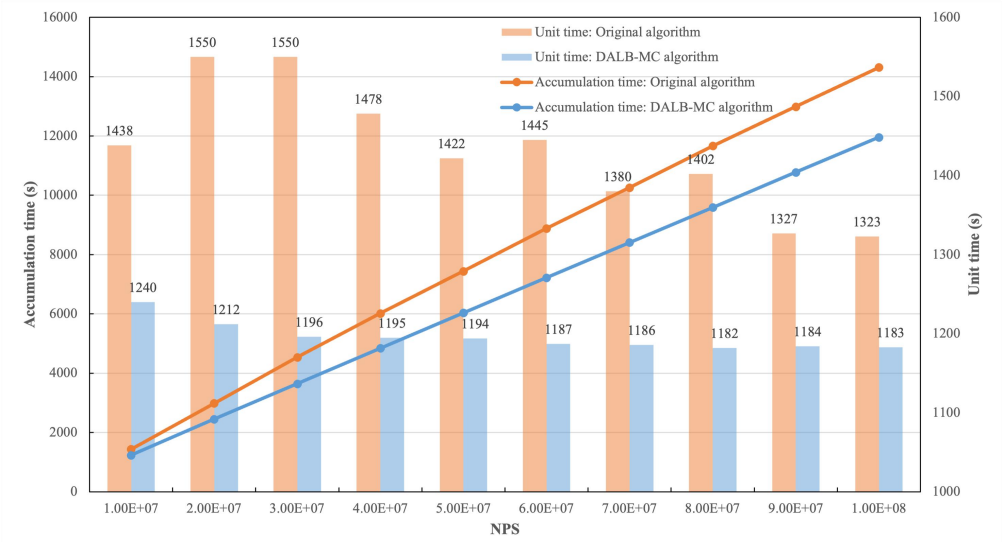


Figure 3.4 Comparison of the computation time of MCShield applying two algorithms

Table 3.2 Comparison of the code performance of MCShield with two algorithms

| Number of simulated particles | Original algorithm | | DALB-MC algorithm | | Calculation Time Reduction | FOM Boost Multiplier |
|---|---|---|---|---|---|---|
| | Computation time (s) | FOM ($min^{-1}$) | Computation time (s) | FOM ($min^{-1}$) | | |
| 1.0E+07 | 1438 | 0.43 | 1240 | 0.40 | 13.77% | -6.07% |
| 2.0E+07 | 2988 | 0.39 | 2452 | 0.42 | 17.94% | 8.27% |
| 3.0E+07 | 4538 | 0.40 | 3648 | 0.49 | 19.61% | 24.49% |
| 4.0E+07 | 6016 | 0.37 | 4843 | 0.45 | 19.50% | 23.03% |
| 5.0E+07 | 7438 | 0.36 | 6037 | 0.43 | 18.84% | 18.40% |
| 6.0E+07 | 8883 | 0.35 | 7224 | 0.41 | 18.68% | 17.69% |
| 7.0E+07 | 10263 | 0.34 | 8410 | 0.43 | 18.06% | 25.20% |
| 8.0E+07 | 11665 | 0.36 | 9592 | 0.44 | 17.77% | 21.61% |
| 9.0E+07 | 12992 | 0.37 | 10776 | 0.44 | 17.06% | 20.02% |
| 1.0E+08 | 14315 | 0.38 | 11959 | 0.45 | 16.46% | 18.96% |

Figure 3.2 evidently shows that 99 workers completed varying numbers of tasks; the process with the highest count, worker 42, completed a total of 1107 subtasks, while the lowest, worker 75, completed 781 subtasks. Worker 42 completed 1.5 times as many subtasks as Worker 75. This demonstrates that DALB-MC algorithm effectively utilizes the computational power of each worker, minimizing idle resource waste. Experimental results indicate that as NPS increases, the dynamic task allocation can achieve a stable computation time acceleration effect of nearly 20%. Additionally, the FOM value with dynamic allocation is generally higher than that for static allocation, showing an improvement of about 20% or more. When comparing the simulation times for individual NPS, the dynamic algorithm exhibits significantly more stability than the static one. Thus, although the master process cannot perform particle transport tasks due to its role in monitoring worker status and real-time task assignment, dynamic task allocation maximizes the computational power of all slave processes, leading to faster total simulation times and enhanced

overall program efficiency.

### 3.3 Verification of Performance Improvement in Massively Parallel Computing

To evaluate the optimization of DALB-MC algorithm at a computational scale relevant to real-world engineering applications, performance comparisons were conducted on a large-scale Linux supercomputing platform. The average number of simulated particles per core ($10^7$) was kept constant across all tests. Both static and dynamic task allocation methods were employed, with each test repeated independently five times. The total time measured includes the computational time for particle transport simulations as well as post-processing time for data merging and output generation. Table 3.3 presents the breakdown of total computation and post-processing times as percentages of the total runtime for multiple test sets.

The results demonstrate that the total runtime for DALB-MC algorithm consistently outperforms the static method, with smaller mean squared deviations across the five repetitions (Figure 3.5). Notably, as the number of parallel cores increases, the dynamic method's advantage in reducing total runtime becomes increasingly pronounced. Additionally, when 2560 cores were used, post-processing time constituted 12.03% of the total runtime for the static method. In contrast, DALB-MC algorithm reduced post-processing time to a mere 0.49% of the total runtime—a practically negligible amount—further underscoring the significant performance benefits of the dynamic approach. This highlights DALB-MC algorithm's ability to enhance both the parallel efficiency and stability of Monte Carlo simulations.

Table 3.3 Comparison of simulation times for MCShield applying two algorithms with different numbers of cores

| Cores | Original algorithm | | DALB-MC algorithm | |
|---|---|---|---|---|
| | Total time (s) | Percentage of Post-processing time | Total time (s) | Percentage of Post-processing time |
| 64 | 2676.0 ± 41.6 | 3.05% ± 0.99% | 2661.6 ± 20.0 | 0 ± 0.00% |
| 128 | 2696.8 ± 26.6 | 2.05% ± 1.43% | 2610.8 ± 35.1 | 0 ± 0.00% |
| 256 | 2820.0 ± 183.5 | 3.99% ± 2.18% | 2620.4 ± 24.7 | 0.01% ± 0.02% |
| 320 | 2824.6 ± 204.2 | 6.75% ± 8.04% | 2609.8 ± 27.8 | 0.01% ± 0.02% |
| 640 | 2748.6 ± 20.4 | 4.73% ± 1.62% | 2630.2 ± 6.7 | 0.03% ± 0.02% |
| 1280 | 2873.4 ± 265.9 | 8.99% ± 9.88% | 2620.6 ± 15.3 | 0.15% ± 0.05% |
| 2560 | 2962.0 ± 243.3 | 12.03% ± 8.58% | 2636.6 ± 10.9 | 0.49% ± 0.00% |

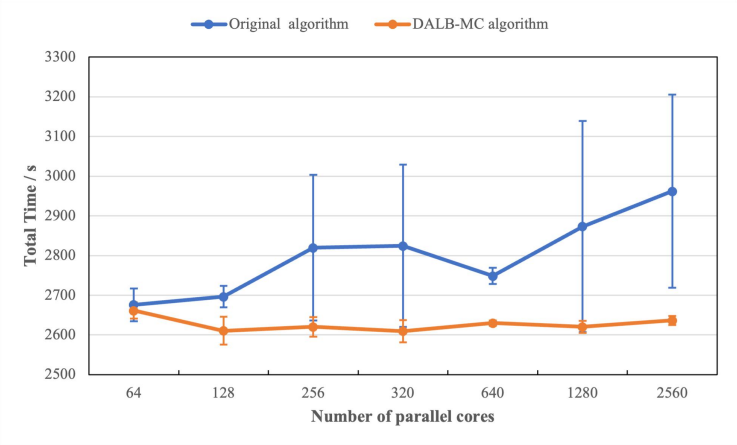


Figure 3.5 Comparison of total time of MCShield applying two algorithms

# 4 Conclusion

The Dynamic Adaptive Load Balancing algorithm for Monte Carlo Codes (DALB-MC algorithm) proposed in this paper addresses the issues of wasted performance and inefficient parallelism in massively parallel particle transport simulations. Unlike traditional static load balancing algorithms commonly employed in Monte Carlo codes, which allocate tasks in a single step, DALB-MC algorithm divides the tasks into multiple discrete subtasks and distributes them adaptively in stages. This approach enhances the stability and efficiency of parallel computing. The correctness and effectiveness of the algorithm were validated through its implementation on MCShield, followed by tests on both small-scale Windows clusters and supercomputing Linux platforms. Experimental results demonstrate that DALB-MC algorithm provides more consistent computational speedup, particularly as the number of simulated particles increases, and significantly improves the overall efficiency of simulation. Additionally, DALB-MC algorithm exhibits superior parallel efficiency in large-scale computing environments, underscoring the necessity of dynamic load balancing in such contexts. However, Since existing Monte Carlo programs exhibit low fault tolerance, future work will focus on developing load-balancing strategies that ensure thread stability and reliability to enable Monte Carlo codes to adapt to changing conditions while maintaining optimal performance.


## Acknowledgements

The authors acknowledge Beijing PARATERA Technology Co., LTD for providing high-performance and AI computing resources for contributing to the research results reported within this paper. URL:http://cloud.paratera.com

## Funding

This work was supported by the National Natural Science Foundation of China (General Program) (U23B2067, U2167209, 12375312, 12175114) and the National Key R&D Program of China (2022YFC2402304).


# Reference


[1] Chandra R. Parallel Programming in OpenMP[M]. Academic Press, 2001.

[2] Brown F B. Recent advances and future prospects for Monte Carlo[J]. 2010.

[3] X-5 Monte Carlo Team, MCNP – A General N-Particle Transport Code, Version 5 – Volume I: Overview and Theory, LA-UR-03-1987, Los Alamos National Laboratory(LANL)(2003).

[4] Shiny, 2013. Load Balancing In Cloud Computing: A Review. IOSR J Comput. Eng. 15(2), 22–29. https://doi.org/10.4018/978-1-7998-1021-6.ch016.

[5] Kaurav, N.S., Yadav, P., 2019. A genetic algorithm based load balancing approach for resource optimization for cloud computing environment. Int. J. Inf. Comput. Sci. 6 (3), 175–184

[6] Alam, M., Ahmad Khan, Z., 2017. Issues and challenges of load balancing algorithm in cloud computing environment. Indian J. Sci. Technol. 10 (25), 1–12. https://doi.org/10.17485/ijst/2017/v10i25/105688

[7] Fatima, S.G., Fatima, S.K., Sattar, S.A., Khan, N.A., Adil, S., 2019. Cloud computing and load balancing. Int. J. Adv. Res. Eng. Technol. 10 (2), 189–209. https://doi.org/10.34218/IJARET.10.2.2019.019.

[8] Kamboj, S., Ghumman, M.N.S., 2016. A Novel Approach of Optimizing Performance Using K-Means Clustering in Cloud Computing. Int. J. Comput. Technol. 15 (14),7435–7443. https://doi.org/10.24297/ijct.v15i14.4942

[9] Haryani, N., Jagli, D., 2014. Dynamic method for load balancing in cloud computing. IOSR J. Comput. Eng. 16 (4), 23–28. https://doi.org/10.9790/0661-16442328

[10] Wang L, Fang W, Du Y. Load Balancing Strategies in Heterogeneous Environments[J]. Journal of Computer Technology and Applied Mathematics, 2024, 1(2): 10-18. https://doi.org/10.5281/zenodo.12599358

[11] Agostinelli S, Allison J, Amako K, et al. Geant4—a simulation toolkit[J]. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 2003, 506(3): 250-303. https://doi.org/10.1016/S0168-9002(03)01368-8

[12] Team X-M C. MCNP - A General Monte Carlo N-Particle Transport Code, Version 5: LA-UR-03-1987, Los Alamos National Laboratory, 2005.

[13] Romano P K, Horelik N E, Herman B R, et al. OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development[J]. Annals of Nuclear Energy, 2015, 82: 90-97. https://doi.org/10.1016/j.anucene.2014.07.048

[14] Böhlen T T, Cerutti F, Chin M P W, et al. The FLUKA Code: Developments and Challenges for High Energy and Medical Applications[J]. Nuclear Data Sheets, 2014, 120(C): 211-214. https://doi.org/10.1016/j.nds.2014.07.049

[15] SCALE A. Comprehensive Modeling and Simulation Suite for Nuclear Safety Analysis and Design[J]. ORNL/TM-2005/39, Version, 2011, 6.

[16] Hirayama H, Namito Y, Bielajew A, et al. The EGS5 code system[M]. 2006.

[17] Leppänen J. A new assembly-level Monte Carlo neutron transport code for reactor physics calculations[C]. M&C 2005, 2005.

[18] S. S. GAO, et al. "Development of a Radiation Shielding Monte Carlo Code: RShieldMC." Proc. Int. Conf. Mathematics & Computational Methods Applied to Nuclear Science and Engineering. (2017)

[19] Wyrzykowski, J. Dongarra, N. Meyer, et al. Parallel Processing and Applied Mathematics[M].

Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, 3911: 228–239.

[20] D. Kranzlmüller, P. Kacsuk, J. Dongarra. Recent Advances in Parallel Virtual Machine and Message Passing Interface[M]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, 3241: 97–104.

[21] Bouteiller A, Cappello F, Herault T, et al. MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging[A]. Proceedings of the 2003 ACM/IEEE conference on Supercomputing[C]. New York, NY, USA: Association for Computing Machinery, 2003: 25.

[22] Thakur R, Rabenseifner R, Gropp W. Optimization of Collective Communication Operations in MPICH[J]. The International Journal of High Performance Computing Applications, SAGE Publications Ltd STM, 2005, 19(1): 49–66.

[23] Schatz F, Koschnicke S, Paulsen N, et al. Master/Slave assignment optimization for high performance computing in an EC2 cloud using MPI[J]. Netw. Protoc. Algorithms, 2012, 4(1): 22-33.

[24] Leenders L. LWR-PVS Benchmark Experiment Venus-3[R]. FCP/VEN/01, SCK/CEN, Mol, Belgium: S.C.K./C.E.N. Nuclear Research Department, 1988.